PATENT

Atty. Dkt. No. 3582:157461

A Messaging Method and Apparatus For Sending and Receiving Messages In A Client Server Environment Over Multiple Wireless Networks

Technical Field

The present invention relates in general to the field of communications and more particularly to messaging between Client device and Servers over multiple wireless Networks that use different access Protocols.

Background

Recent advances in hardware and communication technologies have brought about the era of Client computing over wired and wireless Networks. The proliferation of powerful notebook computers and wireless Client devices promises to provide users with Network access at any time and in any location over various Networks, including the Internet. This continuous connectivity allows users to be quickly notified of changing events, and provides them with the resources necessary to respond in real time even when in transit. For example, in the financial services industry, online traders and financial professionals may be given the power to access mission critical information in real-time, using wireless Client devices.

Conventionally, however, developers of complex, wireless messaging solutions have been forced to develop Applications that are specific to various device types and Network access Protocols in diverse enterprise architectures and platforms. In other words, conventional Client computing solutions have been largely device-specific, Network-specific, or both. For examples, messages may be generated by Client Applications running on a wide variety of Client devices, such as Palm Computing Platform Devices, Windows CE Devices, Paging and Messaging Devices, Laptop Computers, Data-Capable Smart Phones, etc. Depending on the type of Network used by service providers, the Client-generated messages may be transported over Networks having various access Protocols, such as CDPD, Mobitex, Dial-up Internet connections,

CDMA, GSM, ReFlex, etc. As a result, current developers of Client computing solutions must have intimate knowledge of wireless Network characteristics, Protocol environments, wireless links channel characteristics, etc. Therefore, here exists a need to simplify wireless Client and Server Application development environments to support the wide variety of device and Network dependent architectures.

Messaging Application Programming Interface (MAPI) is a messaging architecture and an interface component for Applications such as electronic mail, scheduling, calendaring and document management. As a messaging architecture, MAPI provides a consistent interface for multiple Application programs to interact with multiple messaging systems across a variety of hardware platforms. MAPI provides cross platform support through such industry standards as SMTP, X.400 and Common Messaging Calls. MAPI is also the messaging component of Windows Open Services Architecture (WOSA).

Accordingly, MAPI is built into Windows 95 and Windows NT and can be used by 16-bit and 32-bit Windows Applications. For example, a word processor can send documents and a workgroup Application can share and store different types of data using MAPI. MAPI separates the programming interfaces used by the Client Applications and the service providers. Every component works with a common, Microsoft Windows-based user interface. For example, a single messaging Client Application can be used to receive messages from fax, a bulletin board system, a host-based messaging system and a LAN-based system. Messages from all of these systems can be delivered to a single "universal Inbox."

The Internet community has a well-developed and mature set of layered transport and Network Protocols. Interconnection layer Protocols and interfaces define the specifications for communication between a process or program being executed on one computer and another process or program running on another computer. For example, Transport Control Protocol (TCP) is a transport layer Protocol used to access Applications on other hosts. TCP is a connection-oriented Protocol, a type of transport-layer data communication service that allows a host to send data in a continuous stream to another host. The transport service will guarantee that all data will be delivered to the other end in the same order as sent and without duplication. Communication proceeds

through three well-defined phases: connection establishment, data transfer, and connection release.

The fundamental internetwork service consists of a packet delivery system, and the internet Protocol (IP) defines that delivery. The IP Protocol only specifies the header format, including the source and destination IP addresses; it does not specify the format of the data area. The IP Protocol performs a routing function by choosing a path over which data is sent. Using special procedure called routing Protocols, routers exchange information among themselves and the hosts to which they are connected. This allows them to build tables, called routing tables, which are used to select a path for any given packet from a source to a destination. Although there can be more than one router along the path, each router makes only an individual forwarding decision as to which is the next host or router, i.e., the next Network hop. This method is called hop-by-hop routing and is distinguished from end-to-end Protocol that is implemented at transport through Application layers.

Transmission Control Protocol/Internet Protocol (TCP/IP) are two Protocols that are part of a Protocol suite or family of Protocols layered and designed to connect computer systems that use different operating systems and Network technologies. TCP/IP is a four-layer Protocol suite (the hardware layer is not counted) which facilitates interconnection on the same or different Networks, and in certain Networks such as the Internet, is a requirement for interoperability. Many popular Network Applications have been built directly on top of the TCP over the past decade, making TCP/IP a de-facto Network Access Protocol standard. Another feature offered by the TCP/IP Protocol is Quality of Service (QOS), which allows packets or streams to include QOS requirements. The QOS defines performance properties of a Network service, possibly including throughput, transit delay, priority. The QOS is supported, among other things, by message transmission retries, when the reception of a transmitted message at a destination is not acknowledged.

Another known transport Protocol is known as User Datagram Protocol (UDP), which is a connectionless transport Protocol. The basic data transfer unit is often called a "Datagram" as is well known in the art and is divided into header and data areas. The header contains source and destination addresses and a type field that identifies the

contents of the Datagram. For example, a UDP header consists of a UDP source port and UDP destination port. A UDP message length field indicates the number of octets in a UDP datagram, and a UDP checksum provides an optional checksum of UDP and some parts of the IP header. UDP is a connectionless Protocol, which is also known as a packet switching Protocol. The connectionless Protocol is a data communication method in which communication occurs between hosts with no previous setup. Packets that are sent between two hosts might take different routes, with no guarantee that the packets may reach their destination. Therefore, conventional UDP/IP Protocol does not offer the QOS feature and message retry services offered by the TCP/IP.

Over end-to-end connections, each of the UDP, TCP or IP Protocols have an overhead associated with corresponding headers. For example, the header overhead for TCP is 24 bytes, and for UDP, it is only 8 bytes. The header overhead for IP is 20 bytes (for IP version 4 or IPv4). The IP overhead creates a 44 bytes total overhead for TCP/IP and 20 bytes total over head for UDP/IP. As can bee seen, TCP/IP Protocol is a relatively "chatty" Protocol that requires significantly high transmission overhead. In fact, occasionally, the overhead for TCP/IP is larger than the actual payload itself. The large overhead associated by TCP/IP requires considerable transmission bandwidth, especially over low speed channels, such as, radio frequency (RF) channels used for wireless links.

In summary, none of the existing wireless Protocols provide an end-to-end solution over multiple Networks and multiple Clients. Therefore, in addition to the need for a common architecture through a single, user friendly methodology for providing effective and reliable wireless data solutions for hand-held and laptop computing devices, wireless Networks, and legacy systems, there also exists a need to efficiently and reliably communicate data using minimum bandwidth.

Summary of the Invention

The present invention uses a multi-network transport programming interface that enables Client/Server Applications to be written easily, where such Applications allow Client devices to communicate messages with Server Applications across multiple wireless and wire-line Networks. Moreover, the present invention offers features for

communicating such messages over wireless Networks efficiently, without requiring significant bandwidth, a valuable resource in wireless Networks.

Briefly, according to the present invention, a messaging system for communicating messages in a Client Server environment Over Multiple Wireless Networks support different Network Protocols. In the messaging system of the invention, a Client device executes a Client Application, and a Back-end Server executes a Server Application. A Protocol Gateway encapsulates an underlying Network Protocol of the plurality of wireless Networks. The Client Application and the Server Application communicate messages with each other through the Protocol Gateway independent from the Network Protocol of the wireless Network used for such communication.

As stated above, conventional session based transport mechanisms (e.g. TCP) are designed for LAN based systems with little network latency. These session based transport implementations are extremely chatty and were not designed to consider the amount of bytes sent over the network to maintain the state of that connection. The present invention offers a highly optimized reliable data Transport. The Transport optimizes the over the air communication by utilizing a connectionless send and receive mechanism. In addition, the present invention provides multiple compression mechanisms to reduce the amount of data that needs to be sent over the air. In order to provide a reliable mechanism over a connectionless environment, the Transport provides for message segmentation and reassembly, message retries, and message ACK and NACK service for each supported wireless network. Message segments that are not acknowledged by the peer protocol layer within the configurable time frame are retried automatically by the Transport. The Client or Server Applications do not need to be concerned with segmenting the message and performing message retries. In addition to performing message retries, the Transport supports message duplication detection. To support this reliable mechanism over a connectionless environment, the Transport adds only four bytes to each application message.

The present invention also utilizes a wireless connectivity middle layer gateway, which incorporates a wireless software development environment. This environment insulates a developer from the complexities of the underlying details related to devices and Protocols. The developer works at the Application layer by utilizing a Softer

Development Kit (SDK) that includes software libraries for both Client and Server Application development. The present invention supports solutions and software engineering utilizing technologies such as: Windows NT/95/98, ODBC compliant databases, Palm OS, and Windows CE Client devices, and CDPD, MOBITEX and dial-up Networks.

In this way, wireless technologies and Client devices remain transparent to the data source through the use of Client and Server Application Programming Interfaces (APIs) that support multiple operating environments, for example, Windows 95, 98, CE and NT, UNIX, Palm OS, RIM, etc. These well-defined API's use a set of portable class libraries to aid in rapid Application development. Enterprise support is also available for Internet Proxy Clients, a line optimize access from standard Internet browsers and e-mail Clients. As Client devices and wireless Network technologies evolve, this system will ensure the data solutions are supported.

Brief Description of the Drawings

FIG. 1 is a block diagram of a communication system that advantageously incorporates a messaging system according to the present invention.

FIG. 2 is a block diagram of a HTTP Redirector that interacts with a Web browser and an AIM.*net* Network that is incorporated with the system of FIG. 1.

FIG.4 is a flow diagram numerically depicting a flow of messages that corresponds to an Authentication Challenge Success.

FIG.5 is a flow diagram numerically depicting a flow of messages that corresponds to an Authentication Challenge Failure.

FIG.6 is a flow diagram numerically depicting a flow of messages that corresponds to a Client Application Request to Back-end Server.

FIG.7 is a flow diagram numerically depicting a flow of messages that corresponds to a Back-end Server Response to Client Application.

FIG.8 is a flow diagram numerically depicting a flow of messages that corresponds to a Back-end Server Alert to Client Application.

FIG. 9 shows an OSI Protocol Stack reference model depicting transport interface for interfacing with a PSTN and a CDPD Network using a UDP/IP layers.

FIG. 10 shows OSI Protocol Stack reference model transport for interfacing with a MOBITEX Network via MPAC/MASC layers.

Detailed Description of the Invention

Referring to FIG. 1, a block diagram of a communication system 10 that advantageously incorporates the present invention includes Client devices 12. The Client devices 12 execute corresponding Client Applications, which are developed to provide specific subscriber solutions. For example, service subscribers carrying Palm Pilot Client devices, Windows CE based Client devices or other one-way or two-way messaging Client devices may remain apprised of stock market activities and initiate transactions while roaming within the coverage area of their respective wireless service providers.

As described in detail below, the communication system 10 supports an Intelligent Messaging Network architecture known as Aether Intelligent Messaging (AIM) Network (hereinafter referred to as AIM.*net*) developed by Aether Systems Inc., the assignee of the present Application. AIM.net advantageously incorporates a middleware service in accordance with the present invention that allows for the development of Client and Server Applications independent of the underlying Network Protocols and device configurations. The basic middleware services offered by the AIM.*net* architecture include Client/Server Connectivity, Platform Transparency, Network Transparency, Application Tool Support (, i.e., APIs), Network Management, Interaction With Other Network Services, and Scalability / High Availability.

SYSTEM OVERVIEW

The communication system 10 is configured to support a wide variety of wired and wireless access Network Protocols via an access Network 14. The access Network Protocols include dial-up modem, Analog Cellular, CDPD, Mobitex, Ardis, iDEN, PCS-CDMA or TDMA, GSM, two-way and one-way paging (e.g., ReFlex , Flex, etc.), as well as GEO or LEO satellite Network access Protocols. Depending on the type of a

particular Network, circuit switched or packet switched wireless data transmission Protocols may be used within the system 10. AIM.*net* provides Network transparency to developers of Client and Server Applications. As such, developers do not need to concern themselves with the implementation details of the underlying Network Protocols or with various platform specific encoding, such as big-endian and little-endian.

A number of the Protocol Gateways 16are configured to support a specific Network access Protocol. The Protocol Gateways 16act as an interface between the Network 14 and a wide-area/local-area Networks (WAN/LAN) 18, which is protected from unauthorized access through a firewall 20. Among other things, the WAN/LAN 18 includes one or more Back-end Servers 22 that run Server Applications that communicate messages with Client Applications running on the Client devices 12. Via one or more Aether Message Routers (AMRs) 24, these messages are routed between the Back-end Servers 22 and the Protocol Gateway 16s 16, and other Network components. It should be noted that although the instant specification is described with reference to an specific architecture, a wide variety of wide area and local area Networks (WANs and LANs) that support wired and wireless environments.

The Protocol Gateways 16 are responsible for sending and receiving Application messages between Client Applications and a Back-end Server 22 that supports the Service Type of the Application message, via the AMR 24. For each access Network Protocol that AIM.*net* supports, a corresponding Protocol Gateway 16 supports that Network Protocol. Protocol Gateways 16 communicate directly with one or more AMR 24 using TCP/IP Protocol. In the preferred embodiment of the invention, the Protocol Gateways 16 use clustering for redundancy, scalability and load-balancing of incoming IP traffic across all the nodes within a configured cluster. Under this arrangement, Client Applications are configured to communicate to a single virtual IP address of the Protocol Gateway 16 cluster. This gives the AIM.*net* Network the flexibility to dynamically start and stop the Protocol Gateways 16 without disrupting service. Typically, the Protocol Gateways 16 run outside of the firewall 20. However, the AIM.*net* Architecture does not preclude the Protocol Gateways 16 from running inside an enterprise firewall.

The Back-end Servers 22 and AMRs 24 each have access to corresponding Back-end Server (BES) and Aether AMR 24s (AMR 24) databases 26 and 28,which respectively store message routing and Server Application parameters. The databases 22 and 28 maintain a common pool of information amongst the entire Network Servers. In an exemplary embodiment, this information, which is independent of any specific messaging Application, is stored and accessed from a SQL database.

In order to assist Network administrators manage the AIM.*net* Network; the AIM.*net* architecture incorporates SNMP Server 27 as the mechanism for Network management. Simple Network Management Protocol (SNMP) is a standard Network management Protocol widely used in TCP/IP Networks. By hooking into a customer's SNMP console, the AIM.*net* Network can be easily and effectively managed. In addition to providing SNMP support, AIM.*net* provides Network administrators an internal tool to monitor the health of the Network.

An AIM HTTP Redirector Server 29 enables off the shelf Web browsers to send and receive HTTP requests over the AIM.*net* Network. As described later, the HTTP Redirector 29 works by intercepting HTTP requests from a Web browser and redirecting them over the AIM.*net* Network for fulfillment by an AIM.*net* HTTP Proxy Server. While AIM.*net* provides its own set of advanced services, it also offers support for external enterprise services that might already be in use by an organization. By supporting other vendor services (e.g. security, database, etc.), AIM.*net* can fit into an existing enterprise Networking environment, thereby allowing organizations to utilize their existing Networking environment.

AIM.*net* provides multiple Software Development Kits (SDKs) to assist engineers developing Client and Server Applications. The SDKs contain one consistent set of Application programming interfaces (APIs) and a set of platform specific libraries for all AIM.*net* supported platforms and Networks. In addition to the SDKs, AIM.*net* provides developers a Resource Kit consisting of a set of tools to assist the developers when designing, implementing, and testing their Client and Server Applications.

As described later in detail, AIM.*net* provides a Mobile Client and Server SDK environment to assist engineers developing Client Applications and Back-end Servers 22.

The SDKs provide an easy to use Application Programming Interface (API) and a set of platform specific libraries to perform compression, Network management services, Server to Server communication, Server registration/de-registration, reliable message transport services, etc. Each of the above-described system components supports various types of services that are developed using the SDKs. Some of the provided services are common to all components. While others are Server-specific services.

COMMON NETWORK SERVICES

In an exemplary embodiment, all of the Servers, Protocol Gateways 16, AMR 24, and Back-end Servers 22 utilize Windows NT 4.0 as their operating system to provide a set of common services, including:

- Network Management
- NT Event Logging
- Message Trace Logging
- Run as NT Services
- Server Registration
- Server De-registration
- Server to Server TCP/IP Communication

In a well-known manner, the AIM.*net* Server SDK encapsulates the implementation of these core functions via APIs to insulate Application developers from the hardware, software and Protocol details of the underlying platforms. Provided below is a brief description of the common services.

Network Management Service

All AIM.net Servers support the standard SNMP GET, SET, and GET NEXT operations. In addition, AIM.net Servers can generate SNMP TRAPS for notifying a Network administrator of a critical event. The AIM.net Server SDK provides a common MIB, for basic control and status-handling that is shared by all AIM.net Servers. In addition, the AIM.net Server SDK provides a MIB for each supported Server Type (i.e. Protocol Gateway 16, AMR 24, HTTP Proxy Server, and Back-end Server 22).

Developers developing Back-end Servers 22 can define custom MIBs to support functions specific to their Application needs and register the custom MIBs in a Registered MIBs database 31. Registering a custom MIB with the SNMP MIB Manager is encapsulated by a set of Network management APIs provided by the AIM.net Server SDK.

NT Event Logging Service

All AIM.*net* Servers log critical information (start/stop time, critical errors) to the NT Event log on a corresponding machine on which they are running. Developers developing Back-end Servers 22 can log Application specific events to the NT Event Log via APIs provided by the AIM.*net* Server SDK.

Message Trace Logging Service

All AIM.*net* Servers can optionally log inbound, outbound, and system events on the machine on which they are running. Developers developing Back-end Servers 22 can log Application specific information to an Application-Info-log via APIs provided by the AIM.*net* Server SDK. In this way, developers are not required to know the implementation details of how to log a message to the Inbound, Outbound, or System-Info-logs.

Run as NT Service

In the preferred embodiment of the invention, All AIM.*net* Servers run as NT Services. Rather than having each Server implement the necessary code to run as an NT Service, a Utility program called *AimServiceAny* is used that wraps NT Service functionality around each AIM.*net* Server executable. The benefits of running a Server as an NT Service are as follows:

- ***Automatic Start on Reboot*** - Conventionally, when it becomes necessary to reboot a machine, the user re-booting also log on and manually start any Servers that need to be running on that machine. With an AutoStart function provided by the *AimServiceAny*, each AIM.*net* Server running as an NT Service automatically

restarts *before* the user logs on. This feature is useful if for example the machine reboots at night without human intervention.

- ***No NT Logon Required to Run*** - As an added security measure, AIM.*net* Servers can run without having anyone logged onto the machine and thus prevent unauthorized users from accessing the machine and the Servers.

- ***Network Management Mechanism*** - In addition to SNMP, running as an NT Service provides an additional simple Network management capability by using a remote *SvrMgr* utility provided on all NT Servers to monitor and start/stop AIM.*net* Services running anywhere on the Network.

- ***Startup Dependencies*** - An NT Service can depend on the presence of other Services before it is allowed to start (e.g. some AIM.*net* Servers depend on the fact that SQL Server is running as well as possible Server to Server dependencies).

Server Registration Service

When an AIM.*net* Server is started, it registers itself with the Network by adding an entry to a *RegisteredServers* table in the AIM Database. This enables other AIM.*net* Servers to locate one another on the Network. An API provided by the AIM.*net* Server SDK allows for registering the following Server attributes in the AIM Database:

- Server Class – Protocol Gateway 16, Back-end, and AMR 24

- Server Type - Protocol Gateways 16 types include CDPD, Mobitex and ISP Dialup. Back-end types depend on the Server Application.

- Packet Header Version - Indicates the version of the Packet Header Protocol the Server supports.

- IP Address and Listener Port - Indicates the IP address and the TCP listener port number to be connected to by other Servers in order to communicate with this Server.

Server De-Registration Service

When an AIM.*net* Server is stopped, it de-registers itself from the Network by removing its entry from the *RegisterServers* table in the AIM Database. An API provided by the AIM.*net* Server SDK de-registers a Server in the AIM Database.

Server - to - Server TCP/IP Communications Service

In the preferred embodiment of the invention, AIM.*net* Servers communicate with each other over a TCP/IP socket connection. APIs provided by the AIM.net Server encapsulates the creation, management, and sending/receiving of data over the socket connection.

SERVER-SPECIFIC SERVICES

In addition to the above-described common set of services, each Server also provides additional services that are specific to its functionality. Thus, the AIM.*net* architecture is comprised of core software components that run on:

Protocol Gateway 16;

AMR 24;

Back-end Server 22;

HTTP Proxy Server; and

SNMP MIB Manager.

PROTOCOL GATEWAYS OPERATION AND SERVICES

Using the registration services provided by the AIM.*net* Server SDK, the Protocol Gateways 16 follow a predefined start up sequence to register itself with the AIM.*net* Network. First, each Protocol Gateway 16 adds an entry to the *RegisteredServers* table in the AIM Database and enumerates the list of all AMR 24s 24 registered with the Network in the same domain. Based on the IP Address and Listener Port for each AMR 24, the

Protocol Gateway 16 establishes and manages a TCP/IP connection with each AMR 24 contained in the enumerated list. When a Protocol Gateway 16 connects to an AMR 24, the AMR 24 adds the Protocol Gateway 16 to its *RegisteredServers* cache and begins forwarding messages to the Protocol Gateway 16. If, however, the connection attempt fails, the Protocol Gateway 16 re-attempts to connect to the AMR 24.

GATEWAY PROTOCOL SERVER-SPECIFIC SERVICES

In addition to the above-described common services, the Protocol Gateways 16 are responsible for supporting the following specific series:

Encapsulate the Network Communications Protocol

Each Protocol Gateway 16 encapsulates the underlying Network access Protocol from AMR 24 and Back-end Servers 22. As a result, when the AMR 24 receives a message from a Protocol Gateway 16, it is unaware of the underlying Network access Protocol used for communicating the message.

Message Segmentation

All messages to be transmitted over the Network that exceed a predefined segment size are segmented into multiple message segments.

Message Re-Assembly

All incoming message segments (except the last segment to complete the message) received (including duplicate segments) are immediately acknowledged back to the peer Protocol layer and are queued pending receipt of all message segments via an Inbound Message Map. When the last segment to complete the message is received, the Protocol Gateway 16 does not immediately send an acknowledgment to the peer Protocol layer. Instead, the message segments are assembled into a complete message, which is forward to an appropriate Back-end Server 22 via an AMR 24. When the Back-end Server 22 successfully receives the message, then the Protocol Gateway 16 acknowledges the last segment received thus completing the acknowledgment of the entire message. An

Inbound Message Map manages a separate inbound message map for each unique Link Station ID of a sender.

Message Segment Duplication Detection

When a message segment has been received for a segmented message, the Protocol Gateway 16 checks to make sure the message segment has not been already received (duplicate message segment). If the message segment is a duplicate, it is acknowledged to the peer Protocol layer, discarded and conditionally logged.

Message Duplication Detection

When all message segments have been received for a message, the segments are assembled into a complete message. If the Message ID of the assembled message has been already received (duplicate message), then the message is acknowledged to a corresponding peer Protocol layer, discarded and conditionally logged. Each Protocol Gateway 16 keeps track of the last *n* Message IDs received for each unique Link Station ID.

Message Pacing / Message Retries / Message Time Outs

Any message that is segmented into a number of segments greater than a segmented pacing threshold is sent at a pacing interval, with the threshold and interval being configurable prior to a Gateway Protocol startup. Each Protocol Gateway 16 automatically retransmits any message segment transmitted over the Network that is not acknowledged by a corresponding peer Protocol layer within a configurable number of time. Such Protocol Gateway 16 retries the configured number of times before notifying a Back-end Server 22 that the message could not be delivered to a Client Application.

Forwarding of Ack/Nack Messages

When a message segment is transmitted over the Network, each Protocol Gateway 16 retains knowledge of all outstanding message segments pending acknowledgment (message segments that have not been acknowledged by the peer Protocol layer) via a Pending Acknowledgment Map. The Pending Acknowledgment Map maintains

information pertaining to message segments that have been successfully transmitted and are pending acknowledgment from the peer Protocol layer. If an acknowledgment (positive or negative) is received for a message segment that is not pending acknowledgment, the segment is discarded and conditionally logged.

When all message segments have been positively acknowledged by the peer Protocol layer, the Protocol Gateway 16 sends an ACK control message to the Back-end Server 22 (provided that the Back-end Server 22 has requested such notification) to indicate the message has been successfully delivered to the Client Application. If the number of transmission attempts for the message segment exceeds a configurable number of retry attempts, the Protocol Gateway 16 sends an NACK control message to the Back-end Server 22 to indicate that the message could not be delivered to the Client Application.

AMR OPERATION AND SERVICES

Each AMR 24 communicates to the Protocol Gateways 16 and Back-end Servers 22. Upon start up, the AMR 24 uses the registration services provided by the AIM.*net* Server SDK to register itself with the AIM Network by adding an entry to the *RegisteredServers* table in the AIM Database 28. The AMR 24 also uses the registration services to enumerate the list of all the Protocol Gateways 16 and Back-end Servers 22 that are registered with the AIM Network. Using the IP Address and Listener Port for each Protocol Gateway 16, the AMR 24 establishes and manages a TCP/IP connection with each Protocol Gateway 16 contained in the enumerated list. When a AMR 24 connects to a Protocol Gateway 16, the Protocol Gateway 16 adds the AMR 24 to its *Server Connections* cache and begins to start forwarding messages to the AMR 24. Based on the IP Address and Listener Port for each Back-end Server 22, the AMR 24 also establishes and manages a TCP/IP connection with each Back-end Server 22 contained in the enumerated list. When a AMR 24 connects to a Back-end Server 22, the Back-end Server 22 also adds the AMR 24 to its *Server Connections* cache and can begin to start forwarding messages to the AMR 24.

Each AMR 24 also uses the registration services provided by the AIM.*net* Server SDK to de-register itself from the AIM Network by removing its entry from the

RegisteredServers table in the AIM Database 28. The AMR 24closes the TCP/IP connection with each Protocol Gateway 16. Each Protocol Gateway 16 also removes the AMR 24 from its *Server Connections* cache and immediately stops forwarding messages to the terminating AMR 24. Then, the AMR 24 cleans up any previously allocated resources and terminates.

AMR Server-Specific Services

In addition to the common services that all AIM Servers support, the AMRs 24 are responsible for supporting the following specific services:

AMR Message Authentication Service

The AMR 24 is responsible for determining that the sender of a message is an authorized customer. When the source of a message is a Client device, the AMR 24 uses the device's source address (e.g., IP address or Mobitex MAN number) as the means of identifying authorized access.

When each AMR 24 receives a Client message, it checks the device address against its local cache of authorized devices. If the source address is not found locally, the AMR 24 then checks the AIM Database 28. If the device address is an authorized device, and the customer has permission rights to the requested Service Type, and the requested Service Type is not in use by the customer's account with a different source address, the AMR 24 caches the device address, customer identifier, and requested service type to ensure fast authentication of additional messages from the same source. Then, the message is considered authentic and is forwarded to the proper Back-end Server 22. Each AMR 24 also passes the customer identifier to the Back-end Server 22 to use as a key to search for customer specific information.

In order to support dial-up access, message authentication based on the device's source address is not used, because during a dial-up access, the source address seen by an AMR 24 is the IP Address of the ISP provider. Each subscriber that desires wire-line access has a User ID and Password, which may be selected by the subscriber at the time they subscribe to a service, and saved as part of the AIM Database 28.

Each AMR 24 initially follows the same procedure to authenticate a dial-up message as it does when authenticating a wire-less message. However, in case a message is received from a dial-up connection, the AMR 24 issues an *authentication challenge* to the message source. On receiving the challenge, the Client Application prompts the user to enter their User ID and Password, which are forwarded (encrypted) to the AMR 24 as an authentication request and proceeds with authentication process.

Once a message source has been authenticated, the AMR 24 checks the Service Type and source address of subsequent messages against its authentication cache and allows/disallows the message as appropriate. Preferably, the AMR 24 does not keep the cached mapping between a source address and valid customer indefinitely. A configurable timeout period may be specified, after which cached entries are removed. The timeout interval is the length of time that has passed between success messages from a cached device. When a device times out due to inactivity, the AMR 24 removes it from its cache. For dial-up devices, the AMR 24 also decrements a device's authentication count within the AIM database. The authentication count indicates how many other AMR 24s have heard from the device. When a dial-up device's authentication count drops to zero, the device address is removed from the AIM database 28.

AMR Client Message Routing Service

According to one aspect of the invention, there are two ways in which an AMR 24 can route a Client message to a Back-end Server 22:

1. **Indirect Routing** - Via an *indirect* routing table that maps Message Keys (Service Type and Message ID) to a registered Back-end Server 22 that supports the Message Key.

2. **Direct Routing** - Messages *targeted* at a specific Back-end Server 22.

The form of routing is determined based on the contents of an AIM Message Header. The AIM Message Header is pre-fixed to every Application message. It contains the following fields:

- A 1-byte Server ID that identifies a specific Server of the given Service Type. The value 0 is reserved to indicate that *indirect* routing is desired. A non-zero value indicates that the message is directed at a specific Back-end Server 22.

- A 12-bit Service Type Identifier, which is used by both indirect and direct routing, identifies the type of Service (MarketClip, FX, etc.) associated with the messages.

- A 12-bit Message Identifier that uniquely identifies the message within the context of the specified Service Type required for direct routing.

Indirect Routing

When an AMR 24 receives an incoming message from a Client Application, it checks the Server ID field contained in the AIM Message Header portion of the message. If the Server ID field of the AIM Message Header is zero, the AMR 24 routes the message to the proper Back-end Server 22 by consulting a routing table that maps Message Keys (Service Type and Message ID) to the IP address of one or more connected Back-end Servers 22 .

During Server registration, all Back-end Servers 22 are required to register a list of supported Message Keys. To minimize the number of entries that are made in the routing table, if a given Back-end Server 22 supports the majority of messages for a specific Service Type, it need only register a single *Root* Message Key consisting of only the Service Type. The small subset of Service messages not supported by that Back-end Server 22 would be registered as individual Message Keys by a different Back-end Server 22 of the same Service Type. The AMR 24 always route messages based on the most specific key value (Service Type, Server ID, and Message ID) found in the table. If no specific mapping is found, the AMR 24 uses just the Service Type portion of the key to look for Root message entries. If the AMR 24 locates more than one Back-end Server 22 that satisfies the message key match, it uses round-robin scheduling to pick which the target Back-end Server 22 to rout to.

Consider two Client Applications, MarketClip and FX, Reuters® News Service solutions for real-time reporting on equities and foreign exchanges, with messages for

each Application supported by a corresponding Back-end Server 22. Under this initial configuration, each Application Back-end Server 22 would only have to register its Root Service Type (e.g., MktMon or FX) in order for its Client messages to be routed correctly by the AMR 24. Suppose that both Back-end Servers 22 currently support News requests independently of one another (i.e. there is no common News Back-end Server 22 that both of them use), but a separate News Back-end Server 22 is created to handle ALL News Requests. Ideally, no new software should be sent to service providers so that all future News messages (for either Application) are tagged to go to the new News Server. Rather, the new News Back-end Server 22, upon registration, adds the specific News Message Keys previously handled by the MarketClip and FX Back-end Servers 22 to the AMRs 24 message routing table.

It should be noted that the original Back-end Servers 22 don't need to change because the News Back-end Server 22 Message Keys contain the Service Types and Message ID's specific to the two Applications. Each AMR 24 does its primary routing based on the more specific table entries, the same News messages that would have formerly been routed to the two Back-end Servers 22 , would get routed to the News Back-end Server 22. Thus, the Back-end Servers 22 can be designed around specific Services, rather than a suite of Services that comprise an Application, some of which may be common to other Applications. Under this arrangement, overall response performance improves as specific Services are assigned to their own Back-end Server 22. This is because a Client Application not using a given Service does not have to wait, while the Back-end Server 22 is accessing process requests for a different Service.

Direct Routing

Back-end Servers 22 that maintain state information about a particular Client often require direct routing. For a Client to ensure that a message reaches a specific Back-end Server 22, the AIM Message Header portion of the message contains a non-zero value in the Server ID field. When an AMR 24 sees a non-zero value in the Server ID field, it routes the message to the proper Back-end Server 22 by consulting a routing table that maps Server Keys (Service Type, Message ID, Server ID) to the IP address of a connected Back-end Server 22.

Specifying a Server ID alone is not sufficient to ensure that message is delivered to the proper Back-end. Even when using direct routing, a Back-end Server 22 registers the Service Types and Message IDs it handle; and the Service Type/ Message ID of a direct route message matches those types registered by the Back-end Server 22 with the specified Server ID. Management of Back-end Server 22 IDs is the responsibility of the Application. If an Application runs more than one Back-end with the *same* Server ID, then messages with that Server ID are routed to the Back-end Server 22 whose message routing table contains the most specific match with the messages Service Type and Message ID. If two Back-end Servers 22 map the same Server ID, Service Type, and Message ID, then, as in indirect routing, the AMR 24 uses round robin scheduling to pick a target Back-end Server 22.

A Back-end Server 22 may use both direct and indirect routing on an as needed basis. To illustrate this, consider a Back-end Server 22 that for the most part is stateless, but has one or two logical operations that require several targeted Client/Server messages to complete. If the Back-end Server 22 initiates an operation that requires a targeted response, it places its Server ID in the AIM Message Header portion of the message it sends to the Client Application. When the Client Application responds, it uses the same Server ID in the response message to assure that the response is sent to the original Server. All other "stateless" messages can be sent with a Server ID of 0, so that they are indirectly routed.

AMR Back-end Server Message Routing Service

Back-end Server 22 messages sent to a Client Application first pass through the AMR 24. Each AMR 24 then decides which Protocol Gateway 16 to forward the message to. The AMR 24 chooses the proper Protocol Gateway 16 based on the communications type (UDP, Mobitex, ISP Dialup, etc.) used by a Subscribers Service Provider. The mapping of communication type to Client address is maintained by the AMR 24 based on fixed entries in the AIM Database 28 that maps a Client device's 12 source address to a specific communication type. Each Protocol Gateway 16 also needs to indicate its communication type during the Server registration process. If a Protocol Gateway 16 could not deliver a message to the Client Application, the Protocol Gateway

16 sends a Network control NACK message a the Back-end Server 22 that originated the message, indicating that the message could not be delivered.

Send via ClientDeviceInfo

When a Back-end Server 22 sends a message to a Client Application in response to a received message, the Client device address (referred to, as it's ClientDeviceInfo), which as a part of the received message, is known to the Back-end Server 22. In response, the Back-end Server 22 provides the ClientDeviceInfo as part of the AIMSvrPacket sent to the AMR 24. Consequently, the AMR 24 can then simply pass this information to the appropriate Protocol Gateway 16, which then sends the message to that Client address.

Send via CustomerID

At times, a Back-end Server 22 may need to asynchronously send a message to a subscriber (e.g. MarketClip Alert). Since this message is not in response to an incoming Client message, the ClientDeviceInfo may not be readily available to the Back-end Server 22. Rather than forcing the Back-end Server 22 to keep a mapping between Client identifiers and their LinkStationIDs, a Back-end Server 22 may send a message to a Client based solely on the customer ID. In this case, the AIMSvrPacket sent to a AMR 24 contains a NULL LinkStationID and a valid Client ID. The receiving AMR 24 searches it's authenticated device cache for an active device associated with the specified Client ID and then uses the device's LinkStationID to forward the message to an appropriate Protocol Gateway 16.

BACK-END SERVER OPERATION AND SERVICES

A Back-end Server 22 is an Application specific Server that implements the logic to process messages specific for that type of Server. For example, a FX Back-end Server 22 handles requests related to foreign exchange functions. A Back-end Server 22 communicates directly with one or more AMR 24s. Typically, Back-end Servers 22 run behind the firewall 20. However, the AIM.*net* Architecture does not preclude Back-end Servers 22 from running outside the enterprise firewall 20.

Excluding the Application logic, which may be complex, the development effort to implement a Back-end Server 22 is relatively straightforward. The AIM.*net* Server SDK encapsulate those functions that are common to all Back-end Server 22s, thereby insulating developers from details of transport control, compression, registering and de-registering with the AIM Database 28.

Similar to other Servers, the Back-end Servers 22 uses the registration services provided by the AIM.*net* Server SDK to register themselves with the AIM Network by adding an entry to the *RegisteredServers* table in the AIM Database 28. Each Back End Server 22 establishes a TCP/IP communication with each registered AMR 24, using a corresponding IP Address. When a Back End Server 22 connects to an AMR 24, the AMR 24 adds the Back End Server 22 to its *RegisteredServers* cache and can begin to start forwarding messages to the Back End Server 22. When De-registering itself from the Network, each the Back-end Server 22 removes its entry from the *RegisteredServers* tables in the AIM Database. The Back-end Server 22 notifies each AMR 24 of its impending shutdown. This allows each AMR 24 to remove the Back-end Server 22 from its *RegisteredServers* cache and immediately stop forwarding messages to the terminating Back-end Server 22.

BACK-END SERVER-SPECIFIC SERVICES:

In addition to the common services, the Back-end Servers 22 are responsible for supporting the following specific functions:

Application Protocol Aware Service

From a Back-end Server 22's perspective, it is communicating directly with a Client Application. In reality, however, a Back-end Server 22 is communicating with one or more AMRs 24. In the AIM.*net* architecture, only the Back-end Servers 22 have knowledge of the Application Protocol required to communicate with a Client Application.

Extended AIM.*net* Compression

In the exemplary embodiment, AIM.*net* provides the Adaptive-Huffman base compression service. The AIM.*net* architecture provides the necessary hooks to enable 3rd party OEM compression mechanisms. If a Back-end Server 22 has specific compression requirements for their Application data that is not addressed by AIM.*net* supplied compression services, (i.e. Adaptive-Huffman); the Back-end Server 22 is responsible for providing the compression mechanism.

Security Services

The architecture provides the necessary hooks to enable 3rd party OEM security mechanisms. If a Back-end Server 22 has specific security requirements for their Application data, the Back-end Server 22 is responsible for providing the security mechanism.

Forwarding of Ack/Nack Messages

When a Client message is delivered to the Back-end Server 22, the Back-end Server 22 sends a Network control ACK message to a Protocol Gateway 16 that originally received the message. When the Protocol Gateway 16 receives the Network control ACK message from the Back-end Server 22, it sends a transport level ACK message to the Client's peer Protocol layer indicating that the message was delivered successfully to the Back-end Server 22.

AIM DATABASE

The AIM Database maintains a common pool of information between AIM Servers. This information, which is independent of any specific messaging Application, is stored and accessed from a SQL database known as the AIM Database. The following sections describe the tables that comprise the AIM database schema.

1.1 Schema

1.1.1 ServiceTypes Table

The ServiceTypes table is a list of all the service types supported by AIM.

ServiceTypes Table

Column Name	Type	Description
ServiceName	varchar[30]	Service Name
TypeID	int	ID of the Service
AllowMultiAccess	bit	True if service allows multiple device access from a single user, false if only allows single device access from single user

1.1.2 RegisteredServers Table

The RegisteredServers table is used during the connection process and keeps track of the location and type of all AIM Servers currently running on the Network. Access to this table is through the AIM Server SDK.

RegisteredServers Table

Column Name	Type	Description
DbID	long	Unique DB ID used for cross referencing
ServiceName	varchar[30]	Server Name
Class	int	Server Class e.g. FES, BES, AMR 24 etc
SubClass	int	Server Subclass e.g. CDPD, Mobitex, etc
DeathCount	int	The number of times connecting Servers have failed to connect to the Server
ServerId	byte	Optional ID used for Server-Specific Message Routing
NetHdrVersion	int	Network header version supported by this Server.
IP Address	varchar[15]	Network location of Server
Port	short	Listener port Server monitors for connection requests
PortB	short	A second port the Server monitors
Domain	varchar[20]	Name of the Domain the Server is running in
RegistrationTime	FILETIME	Date/Time when Server registered

1.1.3 ServerMsgMap Table

The ServerMsgMap is accessed during Server Registration, AMR 24 Start-UP and Client Message Routing. This table maps a running AIM Server to the set of Message's that should be routed to that Server. Access to this table is through AIM.*net* Server SDK.

ServerMsgMap Table

Column Name	Type	Description
ServerDBID	long	Cross reference to DbID column in RegisteredServer Table
ServiceType	int	Type of Service message handled by this Server
MessageID	int	Message Identifier of message handled by this Server
ServerID	byte	Optional ID used for Server-Specific Message Routing

1.1.4 AuthorizedUsers Table

The AuthorizedUsers table is accessed during Message authentication. The table contains a list of UserIDs/Passwords with authorized access to the AIM Network. Access to this table is through the AIM Server SDK.

AuthorizedUsers Table

Column Name	Type	Description
UserID	varchar[25]	Identifier chosen by the customer e.g. (rudy, RudyB etc)
Password	varchar[25]	Customer Password
AccountNo	char[8]	Customer Account Identifier
CustomerID	long	Unique CustomerID used for cross referencing

1.1.5 AuthorizedDevices Table

The AuthorizedDevices table is accessed during message authentication. This table contains a list of device addresses with authorized access to the AIM Network. Entries may be permanent (a Mobile Client Device) or temporary (a Wire-line device). Access to this table is through the AIM Server.

AuthorizedDevices Table

Column Name	Type	Description
DevAddress	varchar[25]	Mobile Client device address (IP, MAN, etc)
Wireline	bit	0 = Mobile Client, 1 = Wire-line
CommType	int	Communication Type (CDPD, Mobitex, CDMA, etc) of the Client device
AuthenticationCount	int	No. of AMR 24s currently aware of this device
AccessFlag	int	Used to block access for devices reported missing or stolen
CustomerID	long	Cross reference to Customer ID in AuthenticatedUsers table
Token	long	Token used for security with wireline devices

1.1.6 UserRights Table

The UserRights table is accessed during message authentication. This table contains the service types an authorized user can access. Access to this table is through the AIM Server SDK.

UserRights Table

Column Name	Type	Description
CustomerID	long	Cross reference to CustomerID in AuthenticatedUsers table.
ServiceType	int	Service Type the Customer is authorized to use. Cross reference to TypeID in ServiceType table.

1.1.7 ActiveUsers Table

The ActiveUsers table is accessed during message authentication. This table contains the list of active customer IDs and the services they are using with a count of AMR 24s that have authenticated the account for the service in use. The purpose of the table is to prevent multiple devices from accessing a service with same customer ID. Also, the table contains the LinkStationType and LinkStationID used by the customer so the AMR 24s can support NULL LinkStationID from the Back-end Server 22. Access to this table is through the AIM Server.

ActiveUsers Table

Column Name	Type	Description
CustomerID	long	Cross reference to CustomerID in AuthenticatedUsers table.
ServiceType	int	Service type in use by Account No
AMR 24Count	byte	Number of AMR 24s that have authenticated the account for the service in use

CommType	smallint	Communication Type (CDPD, Mobitex, etc) of the Client device
LinkStationID	varchar[25]	IP/Port or Mobitex Address

1.1.8 CommTypes Table

The CommTypes table is a list of all communication Protocols supported by AIM.

CommTypes Table

Column Name	Type	Description
CommName	varchar[25]	Name of the communication Protocol
TypeID	smallint	Communication Type ID

1.2 Stored SQL Procedures

SQL procedures are used to manage the AIM Database. The following is a list of definitions commonly used as parameters in the stored SQL procedures.

CustomerID – The customer's unique identifier.

UserId – The user Id is used to authenticate ISP dial-up access.

Password – The password is used to authenticate dial-up access.

AccountNo – The account number can be both alpha and/or numeric and is for customer service purposes.

ServiceType – The service type the customer is provisioned to access. For example, MarketClip, MarketTrader, etc.

DeviceAddress - For CDPD devices, this is an IP address in dot notation. For Mobitex, this is a MAN number.

CommType – The type of Network Protocol the Client device is using to access the AIM Network. For example, CDPD, Mobitex, CDMA.

DeviceType – The type of access to AIM Network, either wireless or wireline.

NotifyAMR 24s – True to notify all AMR 24s and false to not notify.

ReturnCode – The return code from the stored procedure.

1.2.1 NewCustomer

This stored SQL procedure allows customer service to enter a new customer using a wireless CDPD device to the AIM Database. User Id and Password are entered as NULL.

Input:

UserID (varchar[25])
Password (varchar[25])
DeviceAddress (varchar[25])
AccountNo (char[8])
ServiceType (int)

Output:

CustomerID (int)

ReturnCode (int) – 0 = Success, 1 = Duplicate User ID, 2 = Duplicate device address.

1.2.2 DeleteCustomer

This stored SQL procedure allows customer service to delete a customer from the AIM Database. This procedure also deletes any devices used by the customer and services provisioned for the customer.

Input:

CustomerID (int)

Output:

ReturnCode (int) – 0 = Success, 1 = Invalid customer id.

1.2.3 AddUser

This stored SQL procedure allows customer service to add a user id and password to the AIM Database.

Input:

UserId (varchar[25])

Password (varchar[25])

AccountNo (char[8])

Output:

CustomerID (int)

ReturnCode (int) – 0 = Success, 1 = Duplicate user id.

1.2.4 DeleteUser

This stored SQL procedure allows customer service to delete a customer from the AIM Database. This procedure also deletes any devices used by the customer and services provisioned for the customer.

Input:

CustomerID (int)

Output:

ReturnCode (int) – 0 = Success, 1 = Invalid customer id.

1.2.5 ChangePassword

This stored SQL procedure allows customer service to change a user's password in the AIM Database.

Input:

UserID (varchar[25])

Password (varchar[25])

Output:

ReturnCode (int) – 0 = Success, 1 = Invalid UserID

1.2.6 AddUserRight

This stored SQL procedure allows customer service to add a user right to a customer defined in the AIM Database.

Input:

CustomerID (int)

ServiceType (int)

Output:

ReturnCode (int) – 0 = Success, 1 = Invalid customer id, 2 = Duplicate entry

1.2.7 DeleteUserRight

This stored SQL procedure allows customer service to delete a user right from a customer defined in the AIM Database.

Input:

CustomerID (int)

ServiceType (int)

Output:

ReturnCode (int) – 0 = Success, 1 = Invalid customer id, 2 = Invalid user right for the customer.

1.2.8 AddDevice

This stored SQL procedure allows customer service to add a device address to a defined customer in the AIM Database.

Input:

DeviceAddress (varchar[25])

Wireline (bit) – 0 = Client, 1 = wireline.

CommType (smallint) – 1 = CDPD, 2 = Mobitex, 3 = ISP Dial up

CustomerID (int)

Token (int)

Output:

ReturnCode (int) – 0 = Success, 1 = Bad parameter, 2 = Duplicate device address, 3 = invalid customer id, 4 = Customer already has device address.

1.2.9 DeleteDevice

This stored SQL procedure allows customer service to delete a device address from a defined customer in the AIM Database.

Input:

DeviceAddress (varchar[25])

Output:

ReturnCode (int) – 0 = Success, 1 = Device address not found

1.2.10 DeleteDeviceByCustID

This stored SQL procedure allows customer service to delete ALL device addresses from a defined customer in the AIM Database.

Input:

CustomerID (int)

Output:

ReturnCode (int) – 0 = Success, 1 = No device address(es) to delete.

1.2.11 SuspendUser

This stored SQL procedure allows customer service to suspend a user and all the user's device address' access to the AIM Network and notify all AMR 24s to remove the device address from it's local cache.

Input:

CustomerID (int)

Output:

ReturnCode (int) – 0 = Success

1.2.12 ReactivateUser

This stored SQL procedure allows customer service to reactivate a user and all the user's device address' access to the AIM Network.

Input:

CustomerID (int)

Output:

ReturnCode (int) – 0 = Success.

1.2.13 SuspendDevice

This stored SQL procedure allows customer service to suspend a device address' access to the AIM Network and notify all AMR 24s to remove the device address from it's local cache.

Input:

DeviceAddress (varchar[25])

NotifyAMR 24s (bit) – True to remove the device address from all AMR 24s memory, false not to.

Output:

ReturnCode (int) – 0 = Success, 1 = Error creating Server Manager, 2 = Error calling Server Manager.

1.2.14 ReactivateDevice

This stored SQL procedure allows customer service to reactivate a device address' access to the AIM Network.

Input:

DeviceAddress (varchar[25])

Output:

ReturnCode (int) – 0 = Success.

1.2.15 GetCustomerID

This stored SQL procedure allows customer service to get the customer identifier associated with a device address.

Input:

DeviceAddress (varchar[25])

Output:

CustomerID (int)

ReturnCode (int) – 0 = Success, 1 = Device address not found.

THE AIM.*net* HTTP PROXY SERVER

Most industry standard browsers support the ability to be configured to access the Internet via a proxy Server instead of communicating directly with an HTTP Web Server. The AIM.*net* HTTP Proxy Server 29 is responsible for handling incoming AIM HTTP requests, sending the request over the Internet to the target Web HTTP Server, and transmitting the response back to the Client device. The AIM.*net* HTTP Proxy Server 29 supports various versions the HTTP Protocol specification. The AIM.*net* HTTP Proxy Server 29 is also responsible for communicating with a target HTTP Web Server. In order to handle each inbound HTTP request, the AIM.*net* HTTP Proxy Server 29 creates and manages a TCP/IP socket connection to the target Web HTTP Server. When the AIM.*net* HTTP Proxy Server 29 receives the response from the Web HTTP Server, it creates an AIM HTTP Response message and formats it for transmission back to the Client device.

AIM HTTP REDIRECTOR

Web browsers typically communicate directly to an HTTP Web Server via TCP/IP. TCP/IP, however, is a chatty LAN Protocol requiring significant overhead that is not cost effective way for browsing the Internet wirelessly. According to one feature of the invention, a HTTP Redirector intercepts raw HTTP requests from a Web browser and redirects the request over the AIM.*net* Network for fulfillment by an AIM.*net* HTTP Proxy Server. When the HTTP Redirector receives a response from the AIM.*net* HTTP Proxy Server, it simply passes the response to the Web browser to process.

Referring to FIG. 2, a block diagram illustrates how the HTTP Redirector interacts with the Web browser and AIM.*net* Network. The HTTP Redirector acts as a "*Client side*" proxy Server allowing it to intercept Web browser HTTP requests. The Web browser still communicates TCP/IP to the HTTP Redirector. However, when communicating over the wireless Network, the HTTP Redirector takes advantage of the optimized wireless Protocol and compression services offered by the AIM.*net* Network. This results in significant byte savings when sending HTTP requests and receiving HTTP responses over a wireless Network. In the exemplary embodiment, the HTTP Redirector

supports Microsoft's Internet Explorer 4.0 and Netscape's Communicator 4.5 Web browsers on the Windows 95, 98, NT, and Windows CE platforms.

SNMP MIB MANAGER

Referring to FIG. 3, the AIM.*net* architecture incorporates SNMP, a Network management Protocol widely used in TCP/IP Networks, as the mechanism for Network administrators to manage the AIM Network. The AIM.*net* SNMP components include a SNMP MIB Manager 30, a SNMP Agent Extension 32, a SNMP.EXE 34, a SNMP Console 36, and a Registered MIBs database 38. As shown, SNMP interfaces with an AIM Server 40 through the SNMP Manager. The SNMP MIB Manager 30 and the SNMP Agent Extension 32 DLL are the AIM.*net* architected components that are responsible for providing the Network management functions for all AIM Servers. Thus, developers are not required to known the SNMP implementation complexities of Windows NT.

The SNMP MIB Manager 30 communicates to the SNMP Agent Extension 32 via a COM interface. The SNMP Extension Agent 32 passes SNMP GET, GETNEXT, and SET operations to the SNMP MIB Manager 30. The SNMP MIB Manager 30 also passes AIM Server generated TRAPS to the SNMP Agent Extension. When the AIM Server 40 is started, it can register one or more Servers defined MIBs with the SNMP MIB Manager 30. For each defined MIB type registered with the SNMP MIB Manager 32, the AIM Server 40 implements a virtual function to support the SNMP GET, GETNEXT, and SET operations for that MIB. Whenever the SNMP Agent Extension 32 passes an SNMP request, the SNMP MIB Manager 30 verifies that the AIM Server 40 has previously registered the MIB at the database 38. If the MIB has been previously registered, the SNMP MIB Manager 30 then calls the Back-end Server 22's virtual function (via COM) for that MIB type.

In addition to the standard SNMP operations described above, the SNMP MIB Manager 30 also supports the capability to communicate an AIM Server generated SNMP TRAP to the SNMP Agent Extension. To generate a SNMP TRAP, an AIM Server 40 uses the services provided by the AIM Server SDK to send the Trap (via COM) to the SNMP MIB Manager 30. The SNMP MIB Manager 30 verifies that the MIB has been

previously registered. If the MIB has been registered, the SNMP MIB Manager 30 passes the Trap to the SNMP Agent Extension 32. The SNMP Agent Extension 32 then passes the SNMP TRAP to the Windows NT SNMP.EXE 34 service executable.

MESSAGE FLOW

The flow of any messages within the Network requires authentication by the AMR 24, via Authentication Challenge success, failures, Client Application Request to Back-end Server 22, Back-end Server 22 Response to Client Application, and Back-end Server 22 Alert to Client Application.

Referring to FIG.4, a flow diagram numerically depicting a flow of messages that corresponds to the Authentication Challenge Success flow. The diagram numerically shows message paths between a Client device 12 and an AMR 24 by numbers 1-8, as follows:

1. The Client Application sends an Application request message to the AMR 24 (the Protocol Gateway 16 is not explicitly involved in Authentication)

2. The Client device may fail the AMR's 24 authentication. There are several ways a Client device can fail authentication. The AMR 24 cannot find the device address in its local cache or the AuthorizedDevices table in the AIM database. The device's security token in the LinkStationID is not the same as the device's security token in the AIM database. The subscriber does not have user rights to the requested service.

3. The AMR 24 sends a NACK message to the Client Application with the appropriate error code.

4. The Client Application then responds with an Authentication Request message consisting of the UserID, secure Password, and the requested Service Type to authenticate.

5. The AMR 24 checks the UserID and Password against the AuthorizedUsers in the AIM database 28. If the UserID/Password are valid, the AMR 24 verifies the subscriber has rights to the requested service. If the subscriber does have user rights to the service, the AMR 24 adds the device address to the AuthorizedDevices table, as well as to its local cache and assigns a security token to the Client device.

6. The AMR 24 sends an Authenticated Response message with a success value to the Client Application to let it know that it has been authenticated. The security token is also sent to the Client device 12.

7. The Client Application then re-sends the original message (step 1) that caused the authentication challenge with the new security token.

8. The AMR 24 verifies the device address against its authentication cache and forwards the message to the proper Back-end Server.

Referring to FIG.5, a flow diagram numerically depicting a flow of messages that corresponds to the Authentication Challenge Success Failure. The diagram numerically shows message paths between the Client device 12 and the AMR 24 by numbers 1-7, as follows:

1. Client Application sends an Application message to the AMR 24 (again, the Protocol Gateway 16 is not explicitly involved in Authentication, although pictured, assume all Client/AMR 24 communications passes through the Protocol Gateway 16).
2. The device fails the AMRs 24 authentication. There are several ways a device can fail authentication. The AMR 24 cannot find the device address in its local cache or the AuthorizedDevices table in the AIM database. The device's security token in the LinkStationID is not the same as the device's security token in the AIM database. The user of the device does not have user rights to the requested service.

3. The AMR 24 sends a NACK message to the Client Application with the appropriate error code.
4. The Client Application then responds with an Authentication Request consisting of the UserID, secure Password and the requested Service Type to authenticate.
5. The AMR 24 checks the UserID and Password against the AuthorizedUsers in the AIM database. The UserID, Password is invalid and/or the user does not have rights to the requested service.
6. The AMR 24 sends an Authentication Response message with a failure value to the Client Application to let it know that the Authentication has failed.
7. The Client may choose to prompt the user to re-enter the UserID and Password and repeat the flow starting from step 4.

Referring to FIG.6, a flow diagram numerically depicting a flow of messages that corresponds to Client Application Request to Back-end Server 22. The diagram numerically shows message paths between the Client device 12 and the AMR 24 by numbers 1-6, as follows:

1. The Client Application creates an Application message and passes the message the Transport layer to transmit over the Network.
2. The Transport layer determines if the message needs to be segmented into multiple segments. The Transport layer then transmits the message over the Network and waits for a transport level ACK.
3. Upon receiving the message, the Protocol Gateway 16 assembles the message segment into a complete Application message (if necessary) and sends the Application message to the next available AMR 24. If no AMR 24 is available, a NACK message is generated by the Protocol Gateway 16 and sent back to the Client Application with the appropriate error code. Preferably, the Protocol Gateway 16 does not immediately send a transport ACK message back to the Client Application. This is done when the Back-end Server 22 receives the Application message and sends an ACK control message back to the Protocol Gateway 16.

4. The AMR 24 looks up the device address and the Service Type (first in its local cache, then if necessary in the AIM Database) to see if the message is from an authorized source. If the message is from an authorized source, the AMR 24 chooses the next available Back-end Server 22 that has registered to support the specified Service Type and then sends the message to that Back-end Server 22. If there are no Back-end Servers 22 registered that support the specified ServiceType, a NACK message is generated by the AMR 24 and sent back to the Client Application with the appropriate error code.
5. Upon receiving the Application message from the AMR 24, the Back-end Server 22 sends an ACK control message back to the Protocol Gateway 16 that received the Application message. The Back-end Server 22 then processes the incoming message.
6. Upon receiving the ACK control message from the Back-end Server 22, the Protocol Gateway 16 sends a transport ACK message to the Client Application.

Referring to FIG.7, a flow diagram numerically depicting a flow of messages that corresponds to Back-end Server 22 Response to Client Application. The diagram numerically shows message paths between the Client device 12 and the AMR 24 by numbers 1-5, as follows:

1. When a Back-end Server 22 responds to a Client Application request, it passes the Response Message, Message Flags, Customer ID and LinkStationID (cached from the previous incoming Request) on the send call. The Message Flags specify whether to compress and/or encrypt the message and whether the Back-end Server 22 requires an ACK message when the Protocol Gateway 16 has successfully delivered the Application message to the Client Application. The Back-end Server 22 then sends the Application message to the next available AMR 24. If no AMR 24 are available, then a false is returned from the send.
2. The AMR 24 uses the LinkStationID to determine the associated communication type (CDPD, Mobitex, etc.) and sends the message to the next available Protocol Gateway 16 of the correct Communication Type.

3. The Protocol Gateway 16 then segments the Application message (if necessary) and transmits the Application message over the Network.
4. The Transport layer receives the message segment and assembles the message segment into a complete Application message (if necessary). The Transport layer then sends a transport ACK message to the Protocol Gateway 16 that sent the message.
5. When the Protocol Gateway 16 receives the transport ACK from the Client Application, it sends an ACK control message back to the Back-end Server 22 that was the source of the original message (if required).

Referring to FIG.8, a flow diagram numerically depicting a flow of messages that corresponds to Back-end Server 22 Alert to Client Application. A Back-end Server 22 unsolicited alert to Client Application flow differs slightly from a Back-end Server 22 response to a Client Application flow. The difference being that a Back-end Server 22 is not responding to a specific request and therefore does not know the LinkStationID of the Client Application, however it does know the Customer ID or the Customer ID and the Application's port number. The diagram of FIG. 8 numerically shows message paths between the Client device 12 and the AMR 24 for this situation by numbers 1-5, as follows:

1. When a Back-end Server 22 sends an unsolicited alert to a Client Application, it passes the Alert Message, Message Flags, NULL LinkStationID and customer/Application information on the send call. The customer/Application info can consist of either the Customer ID or the Customer ID and Application port number. The Message Flags specify whether to compress and/or encrypt the message and whether the Back-end Server 22 requires an ACK message when the Protocol Gateway 16 has successfully delivered the message to the Client Application. The Back-end Server 22 then sends the alert message to the next available AMR 24. If no AMR 24s are available, then a false is returned from the send.

2. The AMR 24 uses the customer/Application information to send the message. If the customer/Application information consist of only the Customer ID, then the AMR 24 first searches its local cache and second the ActiveUsers table to obtain the LinkStationID associated with the Customer ID. If the customer/Application consist of both the Customer ID and Application port number then the AMR 24 first searches its local cache and second the first device assigned to the Customer ID in the AuthorizedDevices table to obtain the LinkStationID. The AMR 24 uses the LinkStationID to determine the associated communication type (CDPD, Mobitex, etc.) and sends the message to the next available Protocol Gateway 16 of the correct Communication Type. If the customer/Application information is only the Customer ID and the LinkStationID are not found in the local cache or ActiveUsers table, then the AMR 24 cannot send the outgoing message to the Client. Therefore the AMR 24 sends a customer inactive message back to the Back-end Server 22 that was the source of the outgoing message. If the customer/Application information is both the Customer ID and Application port number, then the message is always sent if a device address is found in the AuthorizedDevices table for the Customer ID.

3. The Protocol Gateway 16 then segments the Application message (if necessary) and transmits the Application message over the Network.

4. The Transport layer receives the message segment and assembles the message segment into a complete Application message (if necessary). The Transport layer then sends a transport ACK message to the Protocol Gateway 16 that sent the message.

5. When the Protocol Gateway 16 receives the transport ACK from the Client Application, it sends an ACK control message back to the Back-end Server 22 that was the source of the original message (if required).

AIM SOFTWARE DEVELOPMENT KITS

Mobile Client SDK

The Mobile Client SDK is comprised of the following set of platform specific libraries. Each of these libraries exports an easy to use Application Programming Interface (API).

- ***AIM Utility Library***
- ***AIM Transport Library***
- ***AIM Security Library***

Under the preferred embodiment of the invention, the Utility library provides compression services. By keeping the Transport library independent from both the compression and security implementation details, new compression and security mechanisms can be added without the knowledge of the Transport library. This eliminates the need to regression test the Transport library, as well as all Application users of the Transport library when adding a new compression or security mechanism. Because the compression and security solutions provided by the Transport library may not meet the need for all AIM enabled Applications, when new Applications are developed, any specific compression or security requirements of such Applications may be accommodated independent from the Transport library individually, on a component basis. By providing wrapper APIs that encapsulate the default implementation of the compression and or security libraries, developers could choose to write to the wrapper APIs, or directly to the compression and or security APIs.

AIM Utility Library

The AIM Utility library provides Applications with worker functions to perform via an easy to use Application program interface (API). The following section summarizes the major functions provided by the AIM Utility library.

I/O Streaming

Provides worker functions to assist developers streaming in and out Application messages. Serial in and out worker functions are provided for most of the common data types supported by the target platform. The streaming worker functions manage the big-endian little endian issues on behalf of the Application.

Compression Mechanism

Applications can optionally encode Application messages prior to transmitting the message to a target destination. If the encode algorithm determines that it is not optimal to encode the message, the message is not encoded. Also, Applications can optionally decode Application messages prior to processing the message. In order to determine if a message needs to be decoded, Applications can check the encode flag contained in the message header.

AIM Message Header

Every Application message is pre-fixed with the AIM Message Header prior to being sent to its target destination. The AIM Utility library provides Applications with worker functions to set / get the contents of the AIM Message Header. It also provides worker functions to serial out and serial in the contents of the AIM Message Header. Applications are not required to know the internal data representation of the AIM Message Header.

AIM Authentication Messages

In order to access the AIM Network via an ISP Dialup connection, the AIM Network requires that the user provide security credentials to identify themselves. The AIM Utility library provides worker functions to build an AIM Authentication Request message. Applications are not required to know the internal data representation of the AIM Authentication request message. Likewise for the AIM Authentication Response message. Worker functions are provided to determine the authentication status of the request.

AIM Transport Library

The Transport library provides reliable, optimized data communication over various wireless Networks, such as the CDPD and Mobitex wireless Networks, as well as ISP Dial Up wire line access to enabled AIM Client Applications via an easy to use Application program interface (API). The following section summarizes the major functions provided by the Mobile Client Transport library.

- ***Designate Target Destination*** - The Client Application can specify the target destination of the machine to receive the message.

- ***Notification of Success/Failure of Transmission*** - The Client Application receives notification of the success or failure of the message transmission. For those platforms that support a multi-threaded environment (e.g. WinCE), the notification mechanism is via an event that the Transport Library asserts. For those platforms that do not support a multi-threaded environment (e.g. Palm OS), the Client Application is required to continuously poll the Transport Library to determine if the message transmission was successful or failed.

- ***Message Segmentation*** - All messages that are greater than the maximum segment size (configurable) are segmented into multiple message segments.

- ***Message Re-Assembly*** - All multi-segmented messages received are re-assembled into a single message prior to presenting the message to the Application.

- ***Message Retries*** - All message segments that are not acknowledged by the peer Protocol layer within the configured time are retried the configured number of attempts before notifying the Application that the message was delivered (acknowledgment) or not (negative acknowledgment).

- ***Configurable Communication Parameters*** - The communication parameters for the Mobile Client Transport library can be configured to tailor the communication behavior. These values can be configured via the registry (WinX platforms) or the Preferences database (Palm OS platforms) prior to opening the Mobile Client Transport library.

- ***Duplicate Message Segment Detection*** - All duplicate message segments received by the Mobile Client Transport library are acknowledged back to the peer Protocol layer, discarded, and conditionally logged.

- ***Duplicate Message Detection*** - All duplicate messages received by the Mobile Client Transport library are acknowledged back to the peer Protocol layer, discarded, and conditionally logged.

A layered architecture is used for developing the Transport Library. Under this arrangement, each layer (excluding the bottom), which layer encompasses certain functions, requests services from the layer directly below it, and each layer (excluding the top) provides services to the layer directly above it. In order to do its job; layer N employs the services of layer N-1. The division of the Network into logical layers allows a higher level to make use of the more primitive functions of lower levels, without concerning itself with the implementation details of those primitives.

In order to facilitate the request and provision of services, the interfaces between layers are clearly defined for peer-to-peer communication between corresponding layers of both sides of a connection. This allows two machines to specify how they communicate with one another on a level-to-level basis, rather than having to negotiate one giant Protocol for the entire Network.

Transport Layers

Referring to FIG.9 and FIG. 10, the Transport library is represented using the Open Systems Interconnect (OSI) seven-layer communications Protocol stack reference model. For implementing the present invention, layers 5, 6, and 7 are customized as

shown by the diagram of FIG.9, which details the Protocol layers provided by the Transport library.

API Layer

This layer represents layer 7 of the OSI Protocol Stack reference model. Conceptually, it resides between the Application program above and the Simple Network Transport Layer below. The function of Application Program Interface Layer is to provide an interface into the Transport Library by which Client Applications can send and receive messages across multiple wireless Networks without having knowledge of the communication implementation.

Simple Network Transport Layer

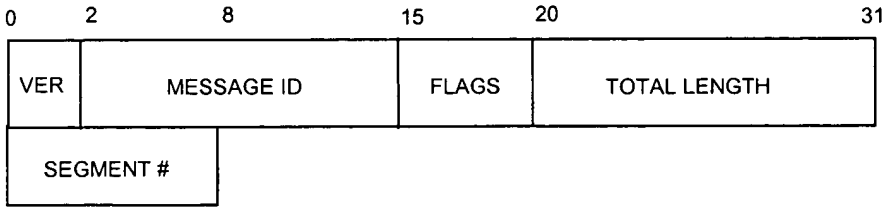
This layer logically represents layer 4 of the OSI Protocol Stack reference model. Conceptually, it resides between the API Layer above and the Communication Layer below. This layer provides the control structure for managing the communications between two communicating Simple Network Transport peer layers. The following sections detail the functions provided by this Protocol layer.

-Message Segmentation

All messages to be sent over the Network that exceed the maximum segment size (configurable) are segmented into multiple message segments. The segment size is configured prior to the Client Application opens the Transport Library. The default maximum segment size is 512 bytes.

-Segment Header

A Transport Header is prefixed to every outbound message segment. The Transport Header is encoded in Network-byte order prior to passing the message segment to the Communication Layer below for transmitting. It is the sole responsibility of the Application to encode any Application specific data in Network-byte order prior to calling the ***AeTransportSend*** API. The diagram below details the Transport Header fields.



SEGMENT HEADER

⇒ **VER**

This field contains the version number of the Segment Header. It consists of two bits, bit 0 and bit 1 of the 1st word in the Segment Header. Valid values are 0 through 3.

⇒ **MESSAGE ID**

This field contains a message identification value. It consists of thirteen bits, bits 2 through 14 of the 1st word in the Segment Header. Valid values are 0 through 8,192. The Simple Network Transport Layer uses the Message ID to discard segment / message duplications and to match acknowledgments with messages.

⇒ **FLAGS**

This field contains Protocol information. It consists of five bits, bits 15 through 19. Valid values are:

Bit 19 - Segmentation Indicator (0 - Message Not Segmented,
1 - Message Segmented)
Bit 18 - Reserved

Bit 17 - Reserved

Bit 16 - Message Type (0 - Positive Acknowledgment,

1 – Negative Acknowledgment)

Bit 15 - Message Indicator (0 - Application Message,

1 - AIM Control Message)

⇒ **TOTAL LENGTH**

This field contains the total number of bytes contained in the message segment to be sent including the Segment Header. It consists of twelve bits, bits 20 through 31 of the 2nd word in the Segment Header. Valid values are 4 through 4,096.

⇒ **SEGMENT NUMBER**

This field identifies the number of this message segment. It consists of 8 bits, bits 0 through 7 of the 3rd word in the Segment Header. Valid values are 2 through 255. The peer Protocol layer uses this number to re-order the message segments into a single complete message. **NOTE:** This field is present only if the Segmentation Indicator is set in the Flags field.

-Notification of Success/Failure of Transmission

When a message segment is passed down to the Communication Layer to be transmitted over the Network, the Simple Network Transport Layer retains knowledge of all outstanding message segments pending acknowledgment (message segments that have not been acknowledged by the peer Protocol layer) via a Pending Acknowledgment Queue. The Pending Acknowledgment Queue maintains information pertaining to message segments that have been successfully transmitted and are pending

acknowledgment from the peer Protocol layer. If an acknowledgment (positive or negative) is received for a message segment that is not pending acknowledgment, the ACK is discarded and conditionally logged.

When all message segments have been positively acknowledged by the peer Protocol layer, the Application is notified (if requested) with a message type of AIM_ACK_MESSAGE and the message ID value associated with the sent message. If the number of transmission attempts for the message segment has exceeded the configured number of retry attempts, the Application is notified with a message type of AIM_NACK_MESSAGE, the message ID value associated with the sent message, and the 2 byte error code containing the reason why the message was not delivered. In order to re-send a message that has been negatively acknowledged, the Application calls a ***AeTransportSend*** API.

-Message Retries

All message segments passed down to the Communication Layer below that are not acknowledged by the peer Protocol layer within the configured time is automatically re-transmitted. The time to wait for an acknowledgment from the peer Protocol layer is configured prior to the Client Application opening the Transport Library. The default time to wait for an acknowledgment from the peer Protocol layer can for example be 15 seconds.

The Simple Network Transport Layer retries the configured number of times before notifying the Application that the message could not be delivered (Negative Acknowledgment). The number of times to retry is configured prior to the Client Application opening the Transport Library. The default number of retry attempts is 3.

-Message Re-Assembly

All incoming message segments received (including duplicate segments) are immediately acknowledged back to the peer Protocol layer and are queued pending receipt of all message segments via the Inbound Message Queues. The Incoming Message Queues manages a separate inbound message queue for each different Link Station ID of the sender.

When all message segments have been received for a message, the segments are assembled into a complete message. If the message ID of the assembled message has been already received (duplicate message), the message is discarded and conditionally logged. This layer keeps track of the last ***n*** Message IDs received for each unique Link Station ID. The number of message IDs to contain in the Message Look Back Queue is configured prior to the Client Application calling ***AeTransportOpen*** to open the Transport Library. The default number of message IDs to maintain in the Message Look Back Queue may be set to 10, for example.

Communication Layer

This layer logically represents layer 4 of the OSI Protocol Stack reference model. Conceptually, it resides between the Simple Network Transport Layer above and the UDP Layer or MPAK Layer below, as shown by FIGs. 9 and 10. This layer is responsible for hiding the Network Protocol interface from the Simple Network Transport layer above. When the Transport Library is opened for CDPD communications, this layer interfaces to the Sockets interface for sending and receiving messages over the CDPD wireless Networks. When the Transport Library is opened for Mobitex communications, this layer interfaces to the MIST layer for sending and receiving MPAKs over the Mobitex wireless Networks.

FIG. 9 shows the multi-network transport programming interface being used for allowing interface with a PSTN and a CDPD Network using a UDP/IP layers. Similarly, FIG. 10 shows the multi-network transport programming interface being used for allowing interface with a MOBITEX Network via MPAC/MASC layers. As shown, the layers below the Communication Layer are Network-specific. This arrangement allows the multi-network transport programming interface of the present invention to be developed to support all of the existing Networks as well as those that may be defined in the future.

AIM Security Library

The Security library provides encryption and decryption services to AIM enabled Applications via an easy to use Application program interface (API). The initial security mechanism is based on Certicon's implementation. The following section summarizes the major functions provided by the Security library.

- ***Key Exchange*** - Public and Private keys are used periodically to establish a common Secret key that both the Client device and Server use when exchanging messages. The reason for this is that the overhead of encrypting using Public/Private keys is much higher than when using a single Secret key. The message flows to securely establish a Secret key between a Client device and a Server is the responsibility of the Security library.

- ***Encryption*** - Mobile Client Applications can optionally encrypt Application messages prior to transmitting the message to the target destination. Messages are encrypted with the secret key negotiated between the Client device and the Server. Encryption is *always* performed after compression.

- ***Decryption*** - Mobile Client Applications can optionally decrypt Application messages prior to processing the message. To determine if a message needs to be decrypted, Applications can check the encrypt flag contained in the message header. Messages are decrypted with the secret key negotiated between the Client device and the Server. Decryption is *always* performed before compression.

Server SDK

The AIM.*net* provides a Server SDK environment to assist engineers developing Protocol Gateway 16s and Back-end Server 22s. The Server SDK is comprised of an easy to use C++ Application programming interface (API) and a set of Windows NT 4.0 libraries. The SDK can be logically divided into the following two categories of classes:

1. AIM Server Classes – These are the core classes that Application developers use when creating new Protocol Gateway 16s and Back-end Server 22s. These classes have no Graphical User Interface (GUI); thereby allowing developers to provide their own custom interfaces.

2. AIM Server User Interface Classes – These classes provide a graphical interface to the AIM Server Classes. Use of these classes is not required when developing a new AIM Server.

AIM Server Classes

The AIM Server Classes are organized in the following simple class hierarchy:

AeServer Class

The AeServer class is the base class for all of the other AIM Server classes and encapsulates those functions that are common to all AIM Servers. These include:

Server Registration/Deregistration – Server subclasses register/de-register from the AIM Database themselves, using methods defined in this class.

Server to Server Connectivity – The logic that determines how two AIM Servers locate and connect to one another is implemented in the AeServer class. The connection flow consists of both establishing a TCP/IP connection as well as the mutual exchange of ServerConnect messages as a means of verifying the identify of each Server.

Server to Server Communication (TCP/IP) - AeServer encapsulates the TCP/IP socket communication between all AIM Servers. AIM Servers can use the communication functions provided by this class to connect, disconnect, send messages, and receive messages over a TCP/IP connection to other AIM Servers. The AIMSvrPacket is the

standard unit of communication between all AIM Servers. The sequence of fields that comprise the AIMSvrPacket is as follows:

AIMSvrPacket Layout

- **Version (4 bits)** - The version number of the AIMSvrPacket.

- **Header Length (4 bits)** - The length of the AIMSvrPacket header in bytes. The AIMSvrPacket header consists of the first 5 fields of the AIMSvrPacket: Version, Header Length, Flags, Total Packet Length and Source Server ID. This length is used by the TCP connection classes to read enough of the packet in order to determine the total size of the rest of the packet.

- **Flags (BYTE)** - contains Protocol information. It consists of eight flag bits, valid values are:

- Bit 1 - Acknowledgment Indicator (1 - Ack Required, 0 - Ack Not Required)
- Bit 2 - Message Type Indicator (1 – Server Connect Message)
- Bits 3-8 – Reserved for future use.

- **Total Packet Length (unsigned long)** - Contains the total number of bytes in the AIMSvrPacket (including the packet header).

- **Source Server Database ID (unsigned long)** - Contains Database ID (a unique value assigned to an AIM Server when the Server registers itself in the AIM database) of the originator of the packet.

- **LinkStationID (variable length)** - Contains the device address of the source or destination of the message contained in the packet. This fields size and content varies depending on the communications type (CDPD, Mobitex, etc) of the device.

- **Message ID (unsigned short)** – Server Packet message identifier.

- **Customer ID (unsigned long)** – AIM Database ID of the customer who owns the device targeted by the message in the Server Packet. Although always present, this field does not always contain a valid value and is set to 0 when not valid. This field is not valid when the AIMSvrPacket contains a Network Control message (Server-to-Server messages independent of Application messages) or when passing a Client message to/from a Protocol Gateway 16 and AMR 24. The primary purpose of the field is for AMR 24 to Back-end communications; to identify the message source on incoming messages, and target a specific customer device on outgoing messages.

- **Port Number (unsigned short)** – Customer Device port number. Although always present in the packet, this field only contains a valid (non-zero) value when a Back-end sends an unsolicited message to a device.

- **AIM Message Header (6 BYTES)** - All Application messages prefix the AIM Message Header to the beginning of the Application message. The AIM Message Header consists of the following fields:

1. **Compression Bits (3-bits)** – 0 = message is not compressed, 1 = AIM supplied compression type, 2 = Aether supplied compression type, 3 = Application supplied compression type.

2. **Security Bits (3-bits)** – 0 = message is not encrypted, 1 = AIM supplied encryption, 2 = Aether supplied encryption, 3 = Application supplied encryption.

3. **Version (3-bits)** – Aether Message header version.

4. **Reserved Bits (7-bits)** – Reserved for future versions.

5. **Service Type (12-bits)** – Identifies which type of Service (MarketClip, FX) the message pertains to. This field is used by both *indirect* and *direct* routing (see section 0)

6. **Message Type (12-bits)** – Uniquely identifies the message within the context of the specified Service Type.

7. **Server ID (1-byte) -** Identifies a specific Back-end Server 22 of the given Service Type. The value of 0 is reserved to indicate that *indirect* routing is desired. A non-zero value indicates that the message is *targeted* at a specific Back-end Server 22.

- **Message Body (variable length)** – Contains the body of the Application message.

AeFEServer Class

The AeFEServer class subclasses AeServer and encapsulates those functions that are common to all Protocol Gateway 16s. All Protocol Gateway 16s inherit from the AeFEServer class. This class performs the following functions on behalf of all Protocol Gateway 16s:

- ***Encapsulates the Transport Header*** - Only this class knows the implementation details of the Transport Header. The Transport Header contains control information for communicating between AIM enabled Client Applications and AIM Protocol Gateway 16s.

- ***Asynchronous Notification of Success/Failure of Transmission*** - This class optionally notifies the original sender of the message indication of the success or failure of the message transmitted to the Client device.

- ***Message Segmentation*** - All outbound messages to be sent to the Client Application that are greater than the maximum segment size (configurable) is segmented into multiple message segments.

- ***Message Re-Assembly*** - All multi-segmented messages received from the Client Application are re-assembled into a single message prior to sending the message to a AMR 24 to route to a registered Back-end Server 22.

    - ***Message Retries*** - All message segments that are not acknowledged by the Client device peer Protocol layer within the configured time is retried the configured number of attempts before notifying the original sender that the message was delivered (acknowledgment) or not (negative acknowledgment).

    - ***Message Pacing***- For large multi-segmented messages, many device modems cannot keep up if they are quickly flooded with a series of segments. Protocol Gateway 16s contain a configurable setting that can be set to slow up the transmission of messages larger than a specified number of segments.

    - ***Duplicate Message Segment Detection*** - All duplicate message segments received from the Client device is acknowledged back to the Client device peer Protocol layer, discarded, and conditionally logged.

    - ***Duplicate Message Detection*** - All duplicate messages received from the Client device is acknowledged back to the Client device peer Protocol layer, discarded, and conditionally logged.

    - ***Configurable Communication Preferences*** - The communication parameters for the Protocol Gateway 16 can be configured to tailor the communication behavior. These values are configured prior to the starting the Protocol Gateway 16.

AeBEServer Class

The AeBEServer class subclasses from AeServer and encapsulates those functions that are common to all Back-end Server 22s. This class performs the following functions on behalf of all Back-end Server 22s:

- ***Generate ACK Control Messages*** - When this class receives an incoming from a Protocol Gateway 16 routed via a AMR 24, it creates an ACK control message and send it back to the originating Protocol Gateway 16 via a AMR 24. When the Protocol Gateway 16 receives this ACK control message, it sends a Transport level ACK message to the Client device that originated the message indicating that the message was delivered to the Back-end Server 22.

- ***Process ACK Control Messages*** - When this class receives an ACK control message from a Protocol Gateway 16, indicating that the Application message was delivered to the Client device, it notifies the derived Back-end Server 22 via a virtual function.

- ***Message Compression/Decompression*** – AeBEServer is responsible for compressing any outgoing messages and decompressing incoming messages. If an AIM provided compression type is involved, compression/decompression is done transparently relative to any subclasses of this type. Alternately, AeBEServer subclasses may implement their own compression algorithms by implementing the appropriate virtual methods that is invoked by AeBEServer at the appropriate times.

- ***Message Encryption/Decryption*** - AeBEServer is responsible for encrypting any outgoing messages and decrypting incoming messages. If an AIM provided encryption type is involved, encryption/decryption is done transparently relative to any subclasses of this type. Alternately, AeBEServer subclasses may implement their own encryption algorithms by implementing the appropriate virtual methods that is invoked by AeBEServer at the appropriate times.

Derived Protocol Gateway 16s

All AIM developed Protocol Gateway 16s inherit from the AeFEServer class. Derived Protocol Gateway 16s provide the following functions:

- ***Encapsulate the Communication Layer*** - Derived Protocol Gateway 16s provide the Network specific interface to the communication layer used by the Protocol Gateway 16 (e.g. UDP sockets for a CDPD gateway, TCP/IP for Mobitex, etc). The parent class (AeFEServer) does not know the implementation details of the underlying communication layer used to send and receive messages to and from Client devices. This is the *sole* responsibility of the derived Protocol Gateway 16.

Derived Back-end Server 22s

All AIM developed Back-end Servers 22 inherit from either the AeBEServer. Derived Back-end Servers 22 provide the following functions:

- ***Process Application Specific Messages*** - All Application specific knowledge is implemented in the derived Back-end Server 22. For example, a News Service can provide Client devices with News stories related to a specific financial entity. The derived News Server's parent class hierarchy (AeBEServer and AeServer) does not know the implementation details of the Application message Protocol. This is the *sole* responsibility of the derived Back-end Server 22.

- ***Special Compression Services*** - If a Back-end Server 22 has specific compression requirements for their Application data that is not addressed by the AIM.*net* supplied compression, the Back-end Server 22 is responsible for providing the compression mechanism.

- ***Special Security Services*** - If a Back-end Server 22 has specific encryption requirements for their Application data that is not addressed by the AIM.*net* supplied encryption, the derived Back-end Server 22 is responsible for providing the encryption mechanism.

Server User Interface Classes

The Server User Interface Class hierarchy parallels the AIM Server Class hierarchy and provides the following types of functionality:

1. Persistent storage of configurable Server settings as well as a common User interface for viewing/editing those settings.

2. Screen based error logging.

3. NT Event Log error logging and automatic batch file error notification.

4. Inbound/Outbound message logging.

5. Inbound/Outbound message statistics.

AeServerApp

AeServerApp is the base class for all of the other AIM Server GUI apps. All AIM Server Apps are complete, windows-based, executable programs. AeServerApp expects its subclasses to provide it with an instance of an AeServer subclass. Of the 5 areas of functionality listed above, AeServerApp provides the following:

1. **Persistent storage of configurable Server settings and common User Interface framework for viewing/editing those settings**. – Persistent storage is implement through the Windows registry and AeServerApp provides the base registry key for all of its subclasses to use. AeServerApp also provides a standard method of viewing/editing Server settings in the form of a PropertySheet. Subclasses provide for their own individual Server settings by adding PropertyPages to the base class

PropertySheet. AeServerApp provides a common page for handling Server settings common to all Server types.

2. **Screen based error logging**. – In addition to providing a window where System events and errors can be displayed, AeServerApp also supplies a separate logging thread that can be used by subclasses when displaying output to their own windows. This thread runs at lower priority then the Server processing threads so that screen logging does not negatively impact performance.

3. **NT Event Log error logging and automatic batch file error notification.** – AeServerApp provides a mechanism whereby system errors can be written to the NT Event log. The level of error reporting is configurable. In addition to the NT Event log, users may specify that a batch file be executed when an errors of a specified severity occur. Such batch files could be used to communicate problems to a System Administrator via email or a pager.

AeFEServerApp

AeFEServerApp derives from AeServerApp and provides the following additional user interface features:

1. **Protocol Gateway specific Server settings** – Provides a User Interface and persistent storage for transport settings such as maximum number of retries, retry timeout interval, segment size, etc.

2. **Inbound/Outbound Message logging** – Provides 2 windows that log each inbound and outbound message. Makes use of the AeServerApp logging thread. Logging may be enabled/disabled for either window.

3. **Protocol Gateway specific statistics** – Gathers and displays statistical totals such as Number of messages sent/received, number of ACKS sent/received.

AeBEServerApp and CBEServerSampleApp

These classes provide a standard GUI for Back-end Server 22s. Both derive from AeServerApp and both provide the same set of user interface features. CBEServerSampleApp came first and was actually written before there was an AeServerApp (although the current version derives from AeServerApp). This difference between the two classes is that CBEServerSampleApp also *derives* from AeBEServer, while AeBEServerApp *has a* AeBEServerApp member (inheritance vs aggregation). Other than that the two classes provide the same set of features:

1. **Inbound/Outbound Message logging** – Provides 2 windows that log each inbound and outbound message. Makes use of the AeServerApp logging thread. Logging may be enabled/disabled for either window.

2. **Back-end specific statistics** – Gathers and displays statistical totals such as Number of messages sent/received, number of ACKS sent/received, and Compressed vs. Uncompressed byte totals.

3. **Application Message Log View** – Provides an additional logging window that Applications should use to log their own errors or trace statements rather than intermingling them with the System messages in the System log window.

AIM WIZARDS AND RESOURCE KIT

In a well-known manner, AIM.*net* can also provides tools that work in conjunction with the Microsoft Visual Developer Studio framework. These tools assist engineers developing Client and Back End Server 22 Applications, as well as stress test and monitor the health of the AIM.*net* Network.

Message Builder Wizard

The AIM.*net* Message Wizard makes it easy for developers to define their Application specific data content of AIM messages. The wizard makes it easier for the developer to focus on adding business value to their Application instead of having to worry about the tedious and error prone task of writing the serialization code to transfer message content between Server and Client. It also automatically generates the code needed to serialize the message content between a Client Application and a Back-end Server 22 Application.

Back End Server 22 App Wizard

The AIM.*net* Back End Server 22 App Wizard makes it easy for developers to create Back End Server 22 Applications. The Back End Server 22 App Wizard generates the Visual Studio C++ project and its associated program and header files to create a Back End Server 22 executable. Back End Server 22 developers would then need to add program logic to support their Application Protocol.

Ping App Wizard

In order to assist engineers developing a Back End Server 22 Ping Application, the AIM.*net* Ping App Wizard makes it easy for developers to create a Ping Back End Server 22 executable. The Ping App Wizard generates the Visual Studio C++ project and its associated program and header files to send an Application defined "*heart beat*" message to a Back End Server 22. Back End Server 22 developers may want to use this tool as a way to monitor the health of their Back End Server 22.

NT Client Simulation Application

In order to assist engineers developing Back-end Server 22s, AIM.*net* also provides a Client Simulation Application. Developers can use the Client Simulation Application to simulate multiple Clients and to generate Back-end Server 22 specific message traffic. The Client Simulation Application supports the following major functions:

- Simulate Up to 256 Clients

- Support Multiple Communication Networks
- ✓ CDPD
- ✓ Mobitex
- ✓ Dial-Up
- ✓ TCP/IP LAN

- Configurable Simulation Attributes
- ✓ Number of messages to send
- ✓ Application defined messages
- ✓ Relative send frequencies for each message type
- ✓ Compression

- Capture / Present Performance Statistics
- ✓ Total messages sent
- ✓ Average message response time

From the forgoing description it may be appreciated that the present invention provides protection against technology obsolescence by supporting seamless integration of information sources with multiple wireless Networks and Client devices. As such, the invention provides a reliable method of data transfer, while optimizing bandwidth constraint of wireless data services and providing end-to-end security. This invention allows for system growth by incorporating the new devices and wireless Network technologies as they become available, without the need to modify Client and Server Applications.

The above-described environment, which has a messaging base architecture, serves as the framework for implementation of the invention. This environment provides Clients/Server connectivity, which provides an enabling mechanism for Application Network connection connectivity. Messaging. Platform transparency is provided by an enabling mechanism for platform independence of the Client devices. Network

transparency is provided by an enabling mechanism for Network independence by hiding the underlining Network Protocol. The SDK provides an easy to use developers tool kit and environment for the design development of each aspect of the Application, the Client device, and Server.